



Procedimentos, Funções e Packages

PROCEDIMENTOS ARMAZENADOS (*STORED PROCEDURES*)

Um procedimento armazenado, também conhecido como *stored procedure*, é um bloco PL/SQL identificado, reutilizável, armazenado como um objeto no banco de dados. Diferentemente dos blocos anônimos, um procedimento pode ter parâmetros de entrada, saída ou entrada/saída. Sua forma geral é:

```
CREATE [OR REPLACE] PROCEDURE nome_procedimento [(
     [nome1 [IN | OUT | IN OUT] tipo1 [DEFAULT valor1]]
     [nome2 [IN | OUT | IN OUT] tipo2 [DEFAULT valor2]] . . .)]
{IS | AS}
   [SEÇÃO de DECLARAÇÃO]
BEGIN
   [SEÇÃO de EXECUÇÃO]
[EXCEPTION]
   [SEÇÃO de TRATAMENTO DE EXCEÇÕES]
END;
```

O comando CREATE PROCEDURE cria a *stored procedure*. Após o nome, há uma relação de parâmetros entre parênteses. Se não houver parâmetros declarados, os parênteses devem ser omitidos.

Cada parâmetro declarado pode ser de entrada (IN), de saída (OUT) ou de entrada e saída (IN OUT). Se não for especificado, o compilador assume como parâmetro de entrada (IN). A execução da *procedure* termina quando é encontrado o comando RETURN ou o END do bloco principal. O exemplo a seguir cria a *stored procedure fatorial_proc*, que retorna o fatorial do número inteiro *n*.



```
CREATE OR REPLACE PROCEDURE fatorial_proc (
    n IN NUMBER,
    fatorial OUT NUMBER
)
IS
BEGIN
    fatorial := 1;
    FOR i IN 2 .. n LOOP
        fatorial := fatorial * i;
    END LOOP;
END;

-- Bloco anônimo que chama a procedure fatorial_proc
DECLARE
    fatorial NUMBER;
BEGIN
    FOR i IN 6 .. 10 LOOP
        fatorial_proc(i, fatorial);
        DBMS_OUTPUT.PUT_LINE('O fatorial de ' || TO_CHAR(i, '99') || ' é '
            || TO_CHAR(fatorial, '999G999G999G999G999'));
    END LOOP;
END;
```



Na *procedure fatorial_proc*, foram declarados 2 parâmetros, um de entrada, *n*, e outro de saída, *fatorial*, que retorna o valor calculado para o bloco que chamou a *procedure*. O resultado, após a execução do bloco anônimo, é mostrado a seguir.

```
O fatorial de   6 é               720
O fatorial de   7 é             5.040
O fatorial de   8 é            40.320
O fatorial de   9 é           362.880
O fatorial de  10 é         3.628.800
```

No exemplo a seguir, a *procedure swap* troca o conteúdo de duas variáveis do tipo VARCHAR2. Ambos os parâmetros são do tipo IN OUT, fornecendo os valores para a *procedure* e retornado o resultado. Observe que não é necessário definir o tamanho do tipo VARCHAR2 para os parâmetros *a* e *b* (na realidade, não é permitido). Isto vale para todos os tipos em que é possível definir o tamanho.

```
CREATE OR REPLACE PROCEDURE swap (
    a IN OUT VARCHAR2,
    b IN OUT VARCHAR2
)
-- Troca os conteúdos de a e b.
IS
    c VARCHAR2(200);
BEGIN
    c := a;
    a := b;
    b := c;
END;

-- Bloco anônimo que chama a procedure swap.
DECLARE
    a VARCHAR2(100);
    b VARCHAR2(100);
BEGIN
    a := 'primeira'; b := 'segunda';
    DBMS_OUTPUT.PUT_LINE('Antes: a = ' || a || ' b = ' || b);
    swap(a, b);
    DBMS_OUTPUT.PUT_LINE('Depois: a = ' || a || ' b = ' || b);
    swap(a, b);
    DBMS_OUTPUT.PUT_LINE('Mais ainda: a = ' || a || ' b = ' || b);
END;
```



O resultado, após a execução do bloco anônimo, é mostrado a seguir.

```
Antes: a = primeira b = segunda
Depois: a = segunda b = primeira
Mais ainda: a = primeira b = segunda
```

Deve-se executar o bloco de criação da *procedure* antes de qualquer referência a ela para que seja armazenada no banco de dados e passe a "existir". Após sua criação, a *procedure* pode ser visualizada no explorador de objetos do *SQL Developer*:

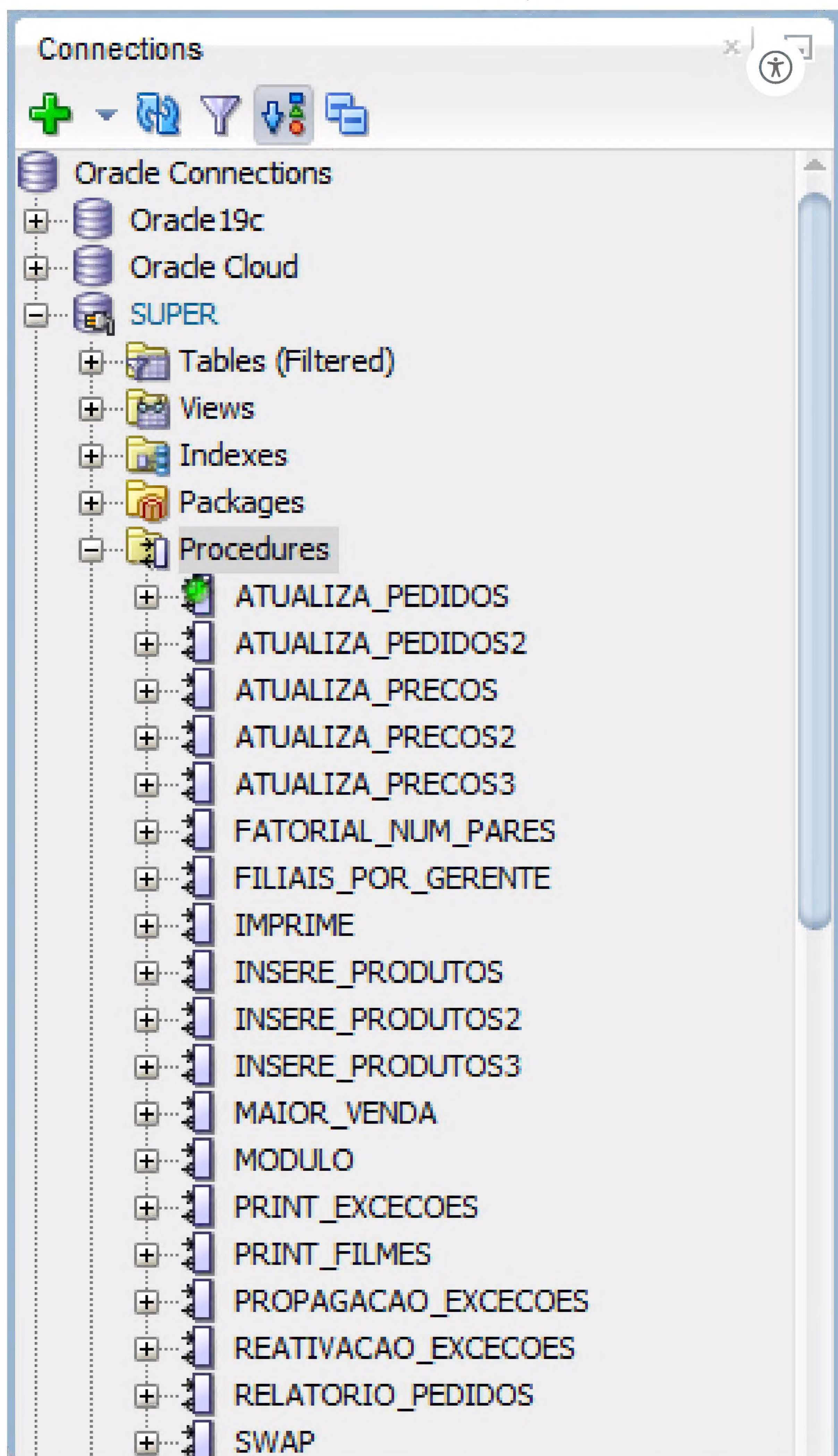
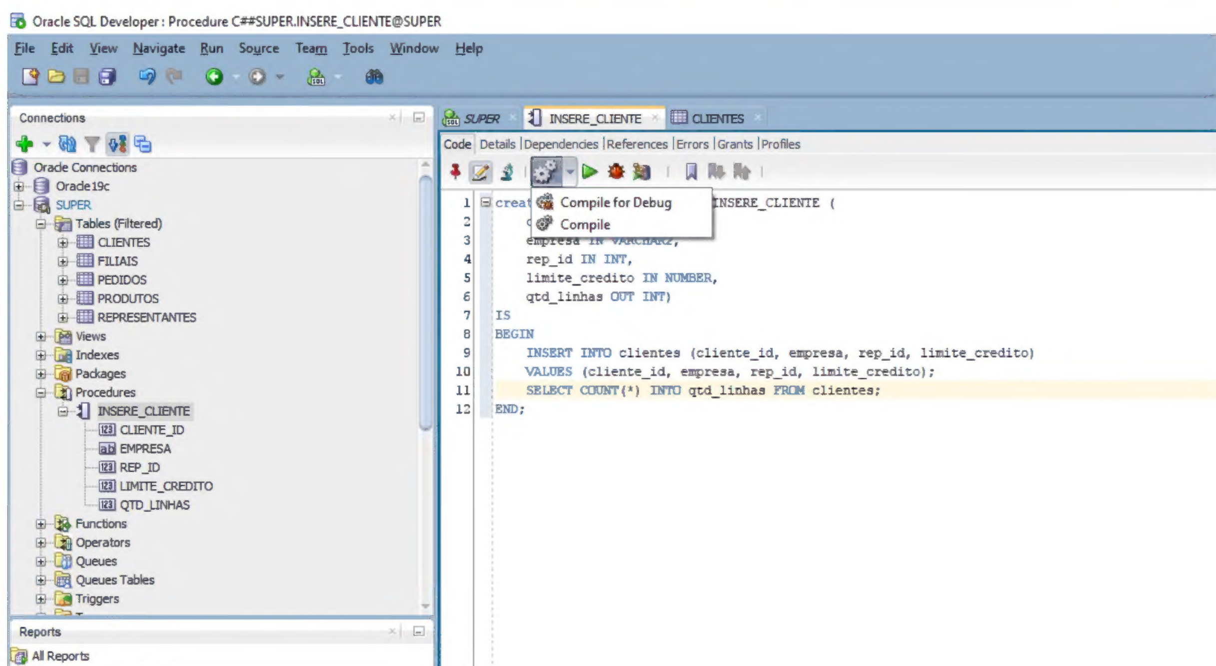

Connections
Oracle Connections
Oracle19c
Oracle Cloud
SUPER
Tables (Filtered)
Views
Indexes
Packages
Procedures
ATUALIZA_PEDIDOS
ATUALIZA_PEDIDOS2
ATUALIZA_PRECOS
ATUALIZA_PRECOS2
ATUALIZA_PRECOS3
FATORIAL_NUM_PARES
FILIAIS_POR_GERENTE
IMPRIME
INSERE_PRODUTOS
INSERE_PRODUTOS2
INSERE_PRODUTOS3
MAIOR_VENDA
MODULO
PRINT_EXCECOES
PRINT_FILMES
PROPAGACAO_EXCECOES
REATIVACAO_EXCECOES
RELATORIO_PEDIDOS
SWAP



Figura 1 – *Procedures*

A alteração de uma *procedure* é simples. Ao clicar sobre o seu nome, no explorador de objetos, uma janela de edição é aberta. Basta fazer as alterações e clicar em *Compile* que as alterações são verificadas e, não havendo erros, a nova versão é armazenada. Na figura a seguir é mostrada a janela de edição do *SQL Developer* com a indicação do comando *Compile*.



Pode-se remover uma procedure de duas formas: através do comando DROP PROCEDURE *nome* ou no explorador de objetos, clicando com o botão direito do *mouse* sobre a *procedure* e selecionando DROP.

Os parâmetros declarados na definição da procedure são chamados de **parâmetros formais**, enquanto aqueles utilizados na chamada da *procedure* são chamados de **parâmetros atuais**. No momento de chamada de uma procedure e antes de sua execução, os parâmetros atuais devem ser mapeados nos parâmetros formais.

Na chamada de uma procedure, os parâmetros atuais podem ser associados aos formais pela posição (referência posicional) ou associando-os explicitamente (referência por nome)

Na referência posicional, a associação é feita pela ordem dos parâmetros atuais na chamada. Nesta forma, a quantidade e os tipos dos parâmetros atuais devem ser iguais aos dos parâmetros formais. Esta é a forma utilizada nos exemplos anteriores.

Na referência por nome, a associação é feita utilizando-se a notação *parâmetro_formal => parâmetro_atual.* A ordem e quantidade de parâmetros atuais não precisam ser iguais às dos formais. Deve-se tomar cuidado, porém, para que sejam atribuídos valores padrão aos parâmetros não obrigatórios. Além disto, não é permitido referências por nome antes de referências posicionais. No exemplo a seguir, o bloco anônimo chama a *procedure fatorial* utilizando as duas formas de passagem de parâmetros.

```
-- Bloco anônimo que chama a procedure fatorial_proc.
DECLARE
    fatorial NUMBER;
BEGIN
    FOR i IN 6 .. 10 LOOP
        -- Todas as quatro formas de chamada estão corretas e são equivalentes
        fatorial_proc(i, fatorial);
        fatorial_proc(n => i, fatorial => fatorial);
        fatorial_proc(i, fatorial => fatorial);
        fatorial_proc(fatorial => fatorial, n => i);
        DBMS_OUTPUT.PUT_LINE('O fatorial de ' || TO_CHAR(i, '99') || ' é '
            || TO_CHAR(fatorial, '999G999G999G999G999'));
    END LOOP;
END;
```

Uma *procedure* pode chamar outras *procedures* e até a si própria. Nesta última situação, a *procedure* é dita recursiva. Deve-se tomar muito cuidado com *procedures* recursivas para que elas não fiquem se chamando indefinidamente. Toda *procedure* recursiva deve ter uma condição de saída.

Considere o fatorial de um número *n*. Pela definição “tradicional”, $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$. Pode-se definir o fatorial de forma recursiva: $n! = n \cdot (n-1)!$. O exemplo a seguir implementa recursivamente a função *fatorial_proc_rec*.

```
CREATE OR REPLACE PROCEDURE fatorial_proc_rec (
    n IN NUMBER,
    fatorial OUT NUMBER
)
IS
BEGIN
    IF n = 2 THEN
        -- 2! = 2
        fatorial := 2;
    ELSE
        -- Calcula (n - 1)!
        fatorial_proc_rec(n - 1, fatorial);
        fatorial := n * fatorial;
    END IF;
END;

-- Bloco anônimo que chama fatorial_proc_rec
DECLARE
    fatorial NUMBER;
BEGIN
    FOR i IN 6 .. 10 LOOP
        -- Todas as quatro formas de chamada estão corretas e são equivalentes
        fatorial_proc_rec(i, fatorial);
        DBMS_OUTPUT.PUT_LINE('O fatorial de ' || TO_CHAR(i, '99') || ' é '
            || TO_CHAR(fatorial, '999G999G999G999G999'));
    END LOOP;
END;
```

O resultado, após a execução do bloco anônimo, é mostrado a seguir.

```
O fatorial de   6 é           720
O fatorial de   7 é         5.040
O fatorial de   8 é        40.320
O fatorial de   9 é       362.880
O fatorial de  10 é     3.628.800
```

FUNÇÕES ARMAZENADAS OU DEFINIDAS PELO USUÁRIO (*STORED FUNCTIONS*)

Uma função definida pelo usuário é muito similar a uma *procedure* exceto pelo fato de que a função retorna um valor. Sua forma geral é:

```
CREATE [OR REPLACE] FUNCTION nome_função (
    [nome1 [IN | OUT | IN OUT] tipo1 [DEFAULT valor1]]
    [nome2 [IN | OUT | IN OUT] tipo2 [DEFAULT valor2]] . . .)
RETURN tipo_retorno
{IS | AS}
   [SEÇÃO de DECLARAÇÃO]
BEGIN
   [SEÇÃO de EXECUÇÃO]
[EXCEPTION]
   [SEÇÃO de TRATAMENTO DE EXCEÇÕES]
END;
```



A declaração de parâmetros e as seções obrigatórias e opcionais são idênticas às de uma *procedure.* O *tipo_retorno* define o tipo do valor retornado e pode ser qualquer tipo válido em PL/SQL. O comando RETURN deve ser utilizado para encerrar uma função, retornando o resultado ao chamador. Um erro ocorre se a execução chegar ao END do bloco principal.

O valor retornado pela função deve ser compatível com o tipo declarado. O compilador PL/SQL faz automaticamente a conversão de tipos, quando possível. Por exemplo, um valor FLOAT retornado por uma função com tipo declarado de retorno NUMBER é automaticamente convertido. Já o retorno de um valor VARCHAR2 por esta mesma função causará erro.

Como nas procedures, os parâmetros podem ser passados por posição, por nome ou combinando as duas formas. O exemplo a seguir implementa a *procedure fatorial_func* através de uma função.

```
CREATE OR REPLACE FUNCTION fatorial_func (
    n IN NUMBER
)
RETURN NUMBER
IS
    fatorial NUMBER := 1;
BEGIN
    FOR i IN 2 .. n LOOP
        fatorial := fatorial * i;
    END LOOP;
    RETURN fatorial;
END;

-- Bloco anônimo que chama fatorial_func_rec
BEGIN
    FOR i IN 6 .. 10 LOOP
        DBMS_OUTPUT.PUT_LINE('O fatorial de ' || TO_CHAR(i, '99') || ' é '
            || TO_CHAR(fatorial_func(i), '999G999G999G999G999'));
    END LOOP;
END;
```



O resultado, após a execução do bloco anônimo, é mostrado a seguir.

```
O fatorial de   6 é                720
O fatorial de   7 é              5.040
O fatorial de   8 é             40.320
O fatorial de   9 é            362.880
O fatorial de  10 é          3.628.800
```

Assim como ocorre com as *procedures*, as funções também podem ser recursivas. O exemplo a seguir apresenta a função *fatorial_func_rec*, que calcula o fatorial de um número inteiro, implementada através de uma função recursiva.

```
CREATE OR REPLACE FUNCTION fatorial_func_rec (
    n IN NUMBER
)
RETURN NUMBER
IS
BEGIN
    IF n = 2 THEN
        RETURN 2;
    ELSE
        RETURN n * fatorial_func_rec(n - 1);
    END IF;
END;

-- Bloco anônimo que chama fatorial_func_rec
BEGIN
    FOR i IN 6 .. 10 LOOP
        DBMS_OUTPUT.PUT_LINE('O fatorial de ' || TO_CHAR(i, '99') || ' é ' ||
            TO_CHAR(fatorial_func_rec(i), '999G999G999G999G999'));
    END LOOP;
END;
```

O resultado, após a execução do bloco anônimo, é mostrado a seguir.

```
O fatorial de   6 é            720
O fatorial de   7 é          5.040
O fatorial de   8 é         40.320
O fatorial de   9 é        362.880
O fatorial de  10 é      3.628.800
```



Como *procedures* e funções são objetos armazenados no banco de dados, o seu escopo é o banco de dados onde foram definidas. Deve-se tomar cuidado para não utilizar identificadores repetidos (nomes) para *procedures* e funções.

Alteração e exclusão de funções são feitas da mesma forma que em *procedures*, através do explorador de objetos ou através do comando DROP FUNCTION.

O QUE SÃO PACOTES (*PACKAGES*)?

Packages são objetos que agrupam elementos da linguagem PL/SQL, tais como variáveis, constantes, tipos, cursores e subprogramas, que guardam relação entre si. *Packages* se assemelham a módulos de algumas linguagens de programação, como Python. O uso de *packages* oferece ao desenvolvedor inúmeras vantagens:

1. **Modularidade.** *Packages* "empacotam" itens que guardam relação lógica entre si, além de oferecerem uma interface simples ao desenvolvedor. O entendimento da função de cada item de uma *package* é mais simples, agilizando o desenvolvimento de aplicações;

2. **Isolamento de detalhes.** Uma *package* possui duas partes: especificação (interface) e corpo (implementação da lógica). Apenas a sua especificação é exposta aos

desenvolvedores, escondendo detalhes de implementação desnecessários ou mesmo confidenciais;

3. **Objetos globais e persistentes.** Todos os objetos declarados na especificação de uma *package* são globais ao esquema ao qual pertencem. Todos os objetos do esquema podem referencia-los. Além disto, as variáveis e cursores globais de uma *package* têm seus valores preservados ao longo da sessão.

A especificação de uma *package* envolve a declaração de objetos que serão expostos aos desenvolvedores. Nela, apenas aquilo que é necessário saber a respeito dos objetos é declarado. A declaração de *procedures* em uma *package*, por exemplo, só contém a relação de parâmetros (e seus tipos). No exemplo a seguir, é mostrada a declaração de uma *package* que implementa duas funções úteis: *inverte* e *localiza*. A primeira, *inverte*(*texto*), recebe como parâmetro de entrada *texto* do tipo *TP_STRING*, declarado dentro da própria *package*, retornando o seu conteúdo invertido. A diretiva SUBTYPE define um novo tipo a partir de um tipo atômico (no caso VARCHAR2). A segunda função, *localiza*(*padrão*, *texto*, *inicio*, *direção*) retorna a posição da primeira ocorrência de *padrão* em *texto*, a partir da posição *inicio*. Até aí nenhuma novidade. A função pré-definida INSTR() faz exatamente isto. A diferença está no parâmetro *direção*, que define se a busca será feita da esquerda para a direita (padrão nas funções de busca) ou na direção oposta. [1] No próprio corpo da função há vários comentários que descrevem a sua implementação.

```
CREATE OR REPLACE PACKAGE pkg_string_func AS
    -- Declaração de objetos que serão expostos ao mundo externo
    SUBTYPE tp_string IS VARCHAR2(1000);
    -- Protótipos (parâmetros e tipo retornado) de funções e procedimentos expostos ao mundo
    FUNCTION inverte (
        texto IN tp_string
    )
    RETURN tp_string;

    FUNCTION localiza (
        padrao  IN tp_string,
        texto   IN tp_string,
        inicio  IN NUMBER,
        direcao IN BOOLEAN
    )
    RETURN tp_string;
END pkg_string_func;
```



Observe que, na especificação da *package*, apenas as informações para chamada de funções e os tipos declarados estão disponíveis. Com estas informações, é possível aos desenvolvedores escreverem programas que utilizam estes objetos.

O corpo da *package* é declarado através do comando CREATE PACKAGE BODY, utilizando-se o mesmo nome utilizado em sua especificação. Nele, a lógica das funções e *procedures* são implementados. No exemplo a seguir, é mostrada a declaração do corpo da *package PKG_STRING_FUNC*.



```
create or replace PACKAGE BODY pkg_string_func AS
    -- Implementação da lógica de funções e procedures

    FUNCTION localiza (
        padrao  IN TP_STRING,
        texto   IN TP_STRING,
        inicio  IN NUMBER,
        direcao IN BOOLEAN
    )
    RETURN
        NUMBER
    IS
        pdr  TP_STRING := padrao;
        txt  TP_STRING := texto;
        ini  NUMBER := inicio;
        tam_pdr NUMBER := LENGTH(padrao);
        tam_txt NUMBER := LENGTH(texto);
        ind     NUMBER;
        achou   BOOLEAN := FALSE;
    --
    -- Procura padrao (o que procurar) em texto (onde procurar) a partir
    -- da posição início. Direcao define se a busca será da esquerda para
    -- a direita (TRUE) ou da direita para a esquerda (FALSE).
    -- Retorna a posição da primeira ocorrência de padrão em texto.
    -- Condições de erro (retorna -1): tamanho(padrao) + inicio - 1 >
    -- tamanho(texto), inicio <= 0, padrao = ''.
    --
    BEGIN
        -- Verifica condições de erro
        IF (tam_pdr + ini - 1 > tam_txt) OR (ini <= 0) OR (tam_pdr = 0) THEN
            RETURN -1;
        END IF;
        -- Se direcao = FALSE, inverto padrao e texto e ajusto inicio. Procurar
        -- um padrão em uma string da direita para a esquerda é a mesma coisa que
        -- o padrão invertido, na string invertida, da esquerda para a direita.
        -- Só é preciso ajustar a posição de início da busca.
        IF NOT direcao THEN
            pdr := inverte(padrao);
            txt := inverte(texto);
            ini := tam_txt - inicio + 1 - tam_pdr + 1;
        END IF;
        -- Ajusta o início para otimizar a busca.
        ind := LEAST(ini, tam_txt - tam_pdr + 1);
        WHILE NOT achou AND ind <= tam_txt - tam_pdr + 1 LOOP
            IF SUBSTR(txt, ind, tam_pdr) = pdr THEN
                achou := TRUE;
            ELSE
                ind := ind + 1;
            END IF;
        END LOOP;
        -- Ajusta a posição retornada em função da direção da busca
        IF achou AND direcao THEN
            RETURN ind;
        ELSIF achou AND NOT direcao THEN
            RETURN tam_txt - ind + 1 - tam_pdr + 1;
        ELSE
            RETURN 0;
        END IF;
    END;

    FUNCTION inverte (
        texto IN TP_STRING
    )
    RETURN
        TP_STRING
    IS
        i   NUMBER;
        tam NUMBER;
        texto_invertido TP_STRING := '';
```

Observe que não é necessário repetir as declarações de variáveis e tipos, já que estes foram completamente definidos na especificação

da *package*. Tampouco as funções e *procedures* devem ser colocadas na mesma ordem em que aparecem na especificaç

Referências a objetos especificados em uma *package* utilizam a notação de ponto: *nome_package.objeto*.

No bloco anônimo a seguir são utilizados os objetos definidos em *PKG_STRING_FUNC*.

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.inverte('abcdefghiabcdefghi'));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.inverte('Socorram-me subi no onibus em marrocoS' ));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.localiza('abc', 'abcdefghiabcdefghi', 1, TRUE));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.localiza('abc', 'abcdefghiabcdefghi', 2, TRUE));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.localiza('abc', 'abcdefghiabcdefghi', 11, TRUE));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.localiza('abc', 'abcdefghiabcdefghi', 17, TRUE));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.localiza('abc', 'abcdefghiabcdefghi', 1, FALSE));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.localiza('abc', 'abcdefghiabcdefghi', 2, FALSE));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.localiza('abc', 'abcdefghiabcdefghi', 11, FALSE));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.localiza('abc', 'abcdefghiabcdefghi', 17, FALSE));
END;
```

O resultado, após a execução é mostrado a seguir.

```
ihgfedcbaihgfedcba
Socorram me subino on ibus em-marrocoS
1
10
0
-1
1
1
10
-1
```

PERSISTÊNCIA DE VARIÁVEIS EM PACKAGES

Variáveis podem ser declaradas e inicializadas na especificação de *packages*. Diferentemente do que ocorre com variáveis de *procedures* e funções, que são destruídas quando a execução do respectivo programa armazenado termina, variáveis em *packages* são persistentes, ou seja, continuam a existir e mantém seus valores

durante toda a sessão corrente. Esta é uma forma de se declararem variáveis verdadeiramente globais: qualquer programa executado na sessão pode referenciar estas variáveis. A seguir, são declaradas duas variáveis na especificação da *package PKG_STRING_FUNC* para contabilizar o número de vezes que cada função é executada (são mostrados apenas os trechos da especificação e do corpo da *package* que foram alterados).

```
CREATE OR REPLACE PACKAGE pkg_string_func AS
    -- Declaração de objetos que serão expostos ao mundo externo
    SUBTYPE TP_STRING IS VARCHAR2(1000);

    -- Variáveis globais
    chamadas_inverte  NUMBER := 0;
    chamadas_localiza NUMBER := 0;

    -- Protótipos (parâmetros e tipo retornado) de funções e procedimentos expostos ao mundo externo
    ...
END pkg_string_func;

CREATE OR REPLACE PACKAGE BODY pkg_string_func AS
    -- Implementação da lógica de funções e procedures

    FUNCTION localiza (
        ...
    )
    RETURN
        NUMBER
    IS
        ...
    BEGIN
        chamadas_localiza := chamadas_localiza + 1;
        ...
    END;

    FUNCTION inverte (
        texto IN TP_STRING
    )
    RETURN
        TP_STRING
    IS
        ...
    BEGIN
        chamadas_inverte := chamadas_inverte + 1;
        ...
    END;

END pkg_string_func;
```

A seguir, o mesmo bloco anônimo utilizado no exemplo anterior.

```
BEGIN
    pkg_string_func.chamadas_inverte  := 0;
    pkg_string_func.chamadas_localiza := 0;
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.inverte('abcdefghiabcdefghi'));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.inverte('Socorram-me subi no onibus em marrocoS' ));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.localiza('abc', 'abcdefghiabcdefghi', 1, TRUE));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.localiza('abc', 'abcdefghiabcdefghi', 2, TRUE));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.localiza('abc', 'abcdefghiabcdefghi', 11, TRUE));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.localiza('abc', 'abcdefghiabcdefghi', 17, TRUE));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.localiza('abc', 'abcdefghiabcdefghi', 1, FALSE));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.localiza('abc', 'abcdefghiabcdefghi', 2, FALSE));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.localiza('abc', 'abcdefghiabcdefghi', 11, FALSE));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.localiza('abc', 'abcdefghiabcdefghi', 17, FALSE));
    DBMS_OUTPUT.PUT_LINE('inverte: ' || pkg_string_func.chamadas_inverte || '   localiza: '
        || pkg_string_func.chamadas_localiza);
END;
```



A primeira vez em que qualquer objeto de uma *package* é referenciado em uma sessão provoca a sua inicialização. Isto faz com que a *package* seja carregada na memória, suas variáveis sejam inicializadas e seu código de inicialização seja executado (será visto na sequência). Como as duas variáveis globais são inicializadas com 0, os dois primeiros comandos não seriam necessários se a *package* ainda não tivesse sido referenciada na seção. O resultado, após a execução é mostrado a seguir.

```
ihgfedcbaihgfedcba
Socorram me subino on ibus em-marrocoS
1
10
0
-1
1
1
10
-1
inverte: 8   localiza: 8
```

De fato, a *procedure localiza* foi chamada 8 vezes. Mas, no bloco anônimo há apenas duas chamadas *procedure inverte*. Por que o contador apresenta o valor 8? Não se esqueça que *inverte* é chamada dentro de *localiza* quando *direção* = FALSE. E como o contador é incrementado dentro de *inverte*, são contabilizadas todas as chamadas a ela.

OVERLOADING DE *PROCEDURES* E FUNÇÕES



Imagine que se deseje criar uma *procedure* ou função que aceite diferentes quantidades ou tipos de argumentos, mas que implemente basicamente a mesma lógica. A solução imediata seria a criação de diferentes versões, uma para cada tipo/quantidade de parâmetros, porém todas com a basicamente a mesma lógica. O problema surge quando forem necessárias alterações na lógica das várias versões. Simplesmente **todas** elas deverão ser alteradas. A solução este problema é utilizar *overloading*.

Overloading é a possibilidade de se declarar *procedures* ou funções com o mesmo nome, porém com quantidades e/ou tipos de parâmetros diferentes. *Overloading* é permitido dentro de *packages*, mas não para *procedures* ou funções individuais. O compilador escolhe que versão será utilizada em função da quantidade/tipo dos parâmetros atuais. No exemplo a seguir, são definidas 4 funções *soma*, que retornam a soma dos dois parâmetros de entrada, se ambos forem de tipos numéricos, e a concatenação de ambos, caso contrário.

```
CREATE OR REPLACE PACKAGE soma_tudo IS

    FUNCTION soma (

        x IN NUMBER,

        y IN NUMBER

    )

    RETURN NUMBER;


    FUNCTION soma (

        x IN VARCHAR2,

        y IN VARCHAR2

    )

    RETURN VARCHAR2;


    FUNCTION soma (

        x IN NUMBER,

        y IN VARCHAR2
```

```
CREATE OR REPLACE PACKAGE BODY soma[illegible] IS

    FUNCTION soma (

        x IN NUMBER,

        y IN NUMBER

    )

    RETURN NUMBER IS

    BEGIN

        RETURN x + y;

    END;


    FUNCTION soma (

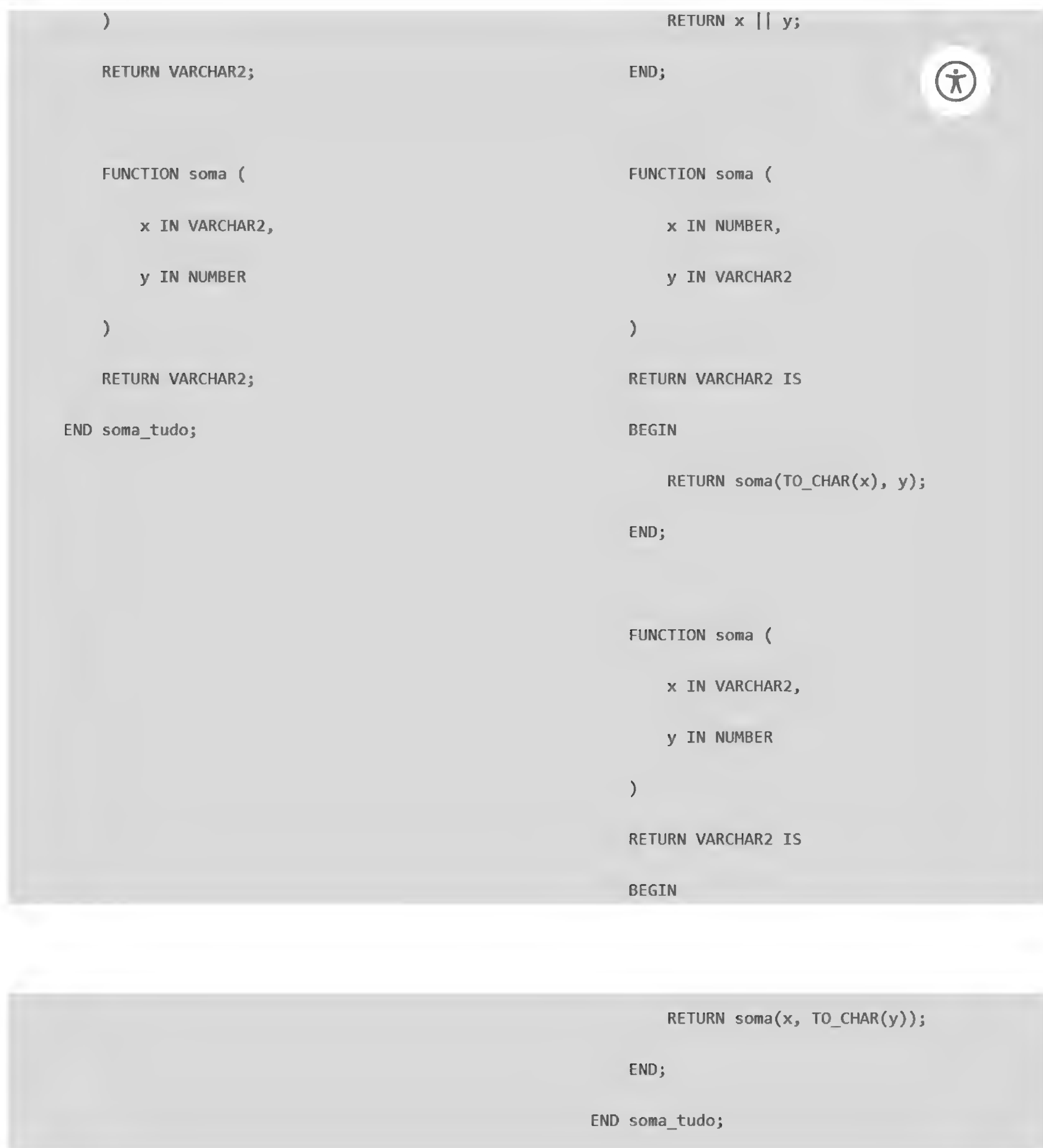
        x IN VARCHAR2,

        y IN VARCHAR2

    )

    RETURN VARCHAR2 IS

    BEGIN
```



```
    )
    RETURN VARCHAR2;

    FUNCTION soma (
        x IN VARCHAR2,
        y IN NUMBER
    )
    RETURN VARCHAR2;
END soma_tudo;
```

```
        RETURN x || y;
    END;

    FUNCTION soma (
        x IN NUMBER,
        y IN VARCHAR2
    )
    RETURN VARCHAR2 IS
    BEGIN
        RETURN soma(TO_CHAR(x), y);
    END;

    FUNCTION soma (
        x IN VARCHAR2,
        y IN NUMBER
    )
    RETURN VARCHAR2 IS
    BEGIN
        RETURN soma(x, TO_CHAR(y));
    END;
END soma_tudo;
```

Observe que as quatro versões possuem parâmetros com tipos diferentes. Observe também que é possível chamar soma de dentro de soma, sem que isto signifique que a função seja recursiva. O bloco anônimo a seguir exemplifica o uso das quatro versões.

```
BEGIN
    DBMS_OUTPUT.PUT_LINE('3 + 4 = ' || soma_tudo.soma(3, 4));
    DBMS_OUTPUT.PUT_LINE('''3'' + ''4'' = ' || soma_tudo.soma('3', '4'));
    DBMS_OUTPUT.PUT_LINE('3 + ''4'' = ' || soma_tudo.soma('3', '4'));
    DBMS_OUTPUT.PUT_LINE('''3'' + 4 = ' || soma_tudo.soma('3', '4'));

END;
```

O resultado, após a execução do bloco anônimo, é mostrado a seguir.



```
3 + 4 = 7
'3' + '4' = 34
3 + '4' = 34

'3' + 4 = 34
```

OBJETOS PRIVADOS E INICIALIZAÇÃO

Packages permitem a criação de objetos privados, visíveis apenas a *procedures* e funções em seu interior. Estes objetos devem ser declarados apenas no corpo da *package*, já que tudo que é declarado em sua especificação é visível externamente.

É possível também criar um trecho de código no corpo da *package* que seja executado uma única vez no momento em que a *package* é referenciada pela primeira vez.

No exemplo a seguir, os dois contadores de chamada serão declarados como privados e seus valores iniciais serão atribuídos pelo código de inicialização. E para que o mundo externo tenha acesso a eles, serão criadas duas funções, *get_cont_inverte* e *get_cont_localiza*, que retornam os valores dos respectivos contadores (no corpo da *package*, são mostradas apenas a declaração das variáveis privadas e a implementação das duas novas funções).

```
CREATE OR REPLACE PACKAGE pkg_string_func AS
    -- Declaração de objetos que serão expostos ao mundo externo
    SUBTYPE TP_STRING IS VARCHAR2(1000);
    -- Protótipos (parâmetros e tipo retornado) de funções e procedimentos expostos ao mundo o
    ...
END pkg_string_func;

CREATE OR REPLACE PACKAGE BODY pkg_string_func AS
    -- Estas duas variáveis são locais à package
    chamadas_inverte  NUMBER := 0;
    chamadas_localiza NUMBER := 0;
    -- Implementação da lógica de funções e procedures
    FUNCTION localiza (
        ...
    FUNCTION get_cont_inverte
    RETURN
        NUMBER
    IS
    BEGIN
        RETURN chamadas_inverte;
    END;

    FUNCTION get_cont_localiza
    RETURN
        NUMBER
    IS
    BEGIN
        RETURN chamadas_localiza;
    END;
END pkg_string_func;
```

TRATAMENTO DE EXCEÇÕES EM *PACKAGES*

Erros de execução podem ocorrer em qualquer parte de um programa, inclusive durante a inicialização de uma *package*. Para isto, é possível incluir uma seção de tratamento de exceções em *packages*, após o código de inicialização. Esta seção capturará todos os erros **exceto** aqueles gerados durante a atribuição de valores padrão a variáveis e constantes. Por isto, é recomendável que toda atribuição de valores iniciais seja feita pelo código de inicialização. Por exemplo, no lugar de *chamadas_inverte NUMBER := 0*, deve-se declarar a variável sem atribuição de valor padrão (*chamadas_inverte NUMBER*) e atribuir o valor padrão depois (*NUMBER := 0*), no código de inicialização. No exemplo a seguir é feita a atribuição de um literal com 3 caracteres a uma variável VARCHAR2(2) em sua declaração. Isto causará um erro não tratado (no corpo da *package*, é mostrada apenas a declaração desta variável e o código de inicialização com a seção EXCEPTION).



```
CREATE OR REPLACE PACKAGE BODY pkg_string_func AS
    -- Estas duas variáveis são locais à package
    chamadas_inverte  NUMBER := 0;
    chamadas_localiza NUMBER := 0;
    vai_dar_erro VARCHAR2(2) :='ERRO';
    -- Implementação da lógica de funções e procedures
    FUNCTION localiza (
        ...
    FUNCTION get_cont_inverte
    RETURN
        NUMBER
    IS
    BEGIN
        RETURN chamadas_inverte;
    END;

    FUNCTION get_cont_localiza
    RETURN
        NUMBER
    IS
    BEGIN
        RETURN chamadas_localiza;
    END;
-- Aqui ficam o código de inicialização e a seção de tratamento de exceções.
BEGIN
    NULL;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Erro durante a inicialização de PKG_STRING_FUNC.');
END pkg_string_func;
```

O bloco anônimo a seguir provocará erro (*ORA-06502 Value Error*) na primeira referência à *PKG_STRING_FUNC*.

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.inverte('abcdefghiabcdefghi'));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.inverte('Socorram-me subi no onibus em marrocoS' ));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.localiza('abc', 'abcdefghiabcdefghi', 1, TRUE));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.localiza('abc', 'abcdefghiabcdefghi', 2, TRUE));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.localiza('abc', 'abcdefghiabcdefghi', 11, TRUE));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.localiza('abc', 'abcdefghiabcdefghi', 17, TRUE));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.localiza('abc', 'abcdefghiabcdefghi', 1, FALSE));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.localiza('abc', 'abcdefghiabcdefghi', 2, FALSE));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.localiza('abc', 'abcdefghiabcdefghi', 11, FALSE));
    DBMS_OUTPUT.PUT_LINE(pkg_string_func.localiza('abc', 'abcdefghiabcdefghi', 17, FALSE));
    DBMS_OUTPUT.PUT_LINE('inverte: ' || pkg_string_func.get_cont_inverte() || '   localiza: ' ||
pkg_string_func.get_cont_localiza());
END;

Error report -
ORA-06502: PL/SQL: numeric or value error: character string buffer too small
ORA-06512: at "C##SUPER.PKG_STRING_FUNC", line 6
ORA-06512: at line 2
```

Já no exemplo a seguir, o erro é capturado.

```
CREATE OR REPLACE PACKAGE BODY pkg_string_func AS
    -- Estas duas variáveis são locais à package
    chamadas_inverte  NUMBER;
    chamadas_localiza NUMBER;
    vai_dar_erro VARCHAR2(2);
    -- Implementação da lógica de funções e procedures
    ...
-- Aqui ficam o código de inicialização e a seção de tratamento de exceções.
BEGIN
    chamadas_inverte := 0;
    chamadas_localiza := 0;
    erro := 'ERRO';
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Erro durante a inicialização de PKG_STRING_FUNC.');
END pkg_string_func;
```



O resultado exibido, após a execução, é mostrado a seguir.

```
Erro durante a inicialização de PKG_STRING_FUNC
ihgfedcbaihgfedcba
Socorram me subino on ibus em-marrocoS
1
10
0
-1
1
1
10
-1
inverte: 8   localiza: 8
```

Pelo fato de o erro ter sido tratado, ele não é propagado e a execução do bloco anônimo pode continuar.

Atividade Extra

A linguagem PL/SQL possui diversas funções pré-definidas (*built in functions*) que são muito úteis no dia-a-dia do desenvolvedor de bancos de dados. Pesquise as funções que são oferecidas, principalmente aquelas de lidam com texto (*strings*) e datas. Se considerar que alguma(s) delas lhe será útil no futuro, inclua-a como um atalho ou *snippet* de código.

Referência Bibliográfica



FEUERSTEIN, S. **Oracle PL/SQL Programming**. 6ª Ed., O'Reilly, 2014.

PUGA, S., FRANÇA, E. e GOYA, M. **Banco de Dados: Implementação em SQL, PL SQL e Oracle 11g**. São Paulo: Pearson, 2014.

Gonçalves, E. **PL/SQL: Domine a linguagem do banco de dados Oracle**. Versão Digital. Casa do Código, 2015.

GROFF, J. R., WEINBERG, P. N. e OPPEL, A. J. **SQL: The Complete Reference.** 3ª Ed., Nova York: McGraw-Hill, 2009.

ELMASRI, R. e NAVATHE, S. B. **Sistemas de Banco de Dados**. 7ª Ed., São Paulo: Pearson, 2011.

[1] Vocês podem pensar que *nunca* precisarão procurar padrões em textos da direita para esquerda. Eu mesmo achava isto até o dia em que precisei desta função e a linguagem que eu estava utilizando não dispunha de uma. Eu mesmo tive que desenvolvê-la.

Ir para exercício